# Exploitation
# of
# Task Level Parallelism

Mayank Mangal


**Department of Computer Science and Engineering**
**National Institute of Technology Rourkela**
**Rourkela-769 008, Odisha, India**

# Exploitation of Task Level Parallelism

*Thesis submitted in partial fulfilment of the requirements for the degree of*

## Master of Technology

*in*

## Computer Science and Engineering

**(Specialization: Software Engineering)**

*by*


## Mayank Mangal

*(Roll No. 212CS3375 )*

*under the supervision of*

## Prof. A. K. Turuk




**Department of Computer Science and Engineering**
**National Institute of Technology Rourkela**
**Rourkela, Odisha, 769 008, India**


**June 2014**

Department of Computer Science and Engineering
**National Institute of Technology Rourkela**
Rourkela-769 008, Odisha, India.

# Certificate

This is to certify that the work in the thesis entitled ***Exploitation of Task Level Parallelism*** by ***Mayank Mangal*** is a record of an original research work carried out by her under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology with the specialization of Software Engineering in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: June 1, 2014

**Dr. A. K. Turuk**
Associate Professor, CSE Department
NIT Rourkela, Odisha

# Author's Declaration

I, **Mayank Mangal** (Roll No. **212CS3375**) understand that plagiarism is defined as any one or the combination of the following

1. Un-credited verbatim copying of individual sentences, paragraphs or illustrations (such as graphs, diagrams, etc.) from any source, published or unpublished, including the Internet sources.

2. Un-credited improper paraphrasing of pages or paragraphs (changing a few words or phrases, or rearranging the original sentence order).

3. Credited verbatim copying of a major portion of a paper (or thesis chapter) without clear delineation of who did or wrote what.

I have made sure that all the ideas, expressions, graphs, diagrams, etc., that are not a result of my work, are properly credited. Long phrases or sentences that had to be used verbatim from published literature have been clearly identified using quotation marks.

I affirm that no portion of my work can be considered as plagiarism and I take full responsibility if such a complaint occurs. I understand fully well that the guide of the thesis may not be in a position to check for the possibility of such incidences of plagiarism in this body of work.

**Place:** NIT Rourkela
**Date:** June 1, 2014

**Mayank Mangal**
Roll: 212CS3375
CSE Department(S/W Engg)
NIT Rourkela, Odisha

# Acknowledgment

I am grateful to numerous local and global peers who have contributed towards shaping this thesis. At the outset, I would like to express my sincere thanks to Prof. A. K. Turuk for his advice during my thesis work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. He has helped me greatly and been a source of knowledge.

I extend my thanks to our HOD, Prof. S. K. Rath for his valuable advices and encouragement.

I am really thankful to my all friends Shakya S Das, Nipun Madan, Pallavi Thummala, Apoorva Sachhan, Lov Kumar, Manish Sachdeva, Amit Pal (SD Group). My sincere thanks to everyone who has provided me with kind words, a welcome ear, new ideas, useful criticism, or their invaluable time, I am truly indebted.

I would also like to thank specially Priyanka Bansal and Deepak Pal(Friend) for standing besides me all the time and support me morally and ethically.

I must acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.

Last, but not the least, I would like to dedicate this thesis to my family, for their love, patience, and understanding.

***Mayank Mangal***
***Roll: 212CS3375***

# Abstract

Existing many systems were supporting task level parallelism usually involving the process of task creation and synchronization. The synchronization of task requires the clear definition about existing dependencies in a program or data-flow restraints among functions(tasks), or data usable information of the tasks. This thesis describes a method called Symbol-Table method which will used to exploits and detects the task level parallelism at inner level of sequential C-programs.

This method is made up of two levels: a normal symbol table and an extended symbol table. A sequential program of C language is the input to the normal gcc compiler in which the procedures are defines as functions(tasks). Than we generate a normal symbol table with specific command by gcc compiler in linux as an output.Then we use the information of that symbol table for generating the extended symbol table with additional information about variable's extended scopes and inner level function dependency. This extended symbol table is generated by the use of previously generated normal symbol table on the basis of variable's scope and L-value/R-value attributes. By that table we can identify the functions and variables those who are sharing the common variables and those who are accessing the different functions with extended scopes respectively. Then we can generate the program dependency graph by the using of that extended symbol table's information with a specific java program. A simple program for using this method has been implemented on a 64-bits linux based multiprocessor.Finally we can generate the Function graph for every variable in the program with the help of table's info and dependency graph's states.From that graph we can get the info about extended scoping of variables to identify and exploit the task level parallelism in the program. Then we can apply the parallelism with MPI or other parallel platforms to get optimized and error free parallelism.

***Keywords:*** *Task synchronization; Function dependencies; Symbol table; Program Dpendency graph;*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

**Motivation**

**Thesis Contribution**

**Project Overview**

**Thesis Organization**

# Chapter 1

# Introduction

## 1.1 Motivation

Parallel programming is much more difficulties and erroneous than sequential programming. It has much more complex outcomes from the requirement to control the interactions among the concurrent tasks or processes. So, More intension is required to confirm that the parallel source programs give the correct outcomes. A currently working parallel source program is not a correct source program always. Likewise, a error-free parallel source program does not always compulsorily results in a good performance.So, to achieve the good performance and good efficiency, a parallel program must have load balance,low overhead and the good data locality. In general, parallel programs are much complex and difficult to maintain because they implement complex type of parallel behavior algorithms, and hold platform-specific optimized source code.

The uninvited difficulties of parallel programming have energized research in the range of parallelizing and rebuilding compilers. These parallelizing compilers consequently discover parallelism in successive projects and rebuild them into parallel projects. The lion's share of parallelizing compilers [1] [2] [3] have concentrated on parallelism inside loops, where this parallelism gives outcome from executing free cycles of a loop in a parallel way. That is usually called as loop level parallelism. In spite of the fact that these type of parallelizing compilers take out the requirement for parallel programming and are for the most part compelling, later studies [4] [5] have indicated that they have some constraints, and that this

parallelism is not sufficient to use the all resources of the parallel computers.

So we finally came to task-level parallelism which provides a task(function) as a procedure invocation,a program block or an arithmetic operation. A few provisions are all the more characteristically communicated as an accumulation of related tasks [6] [5]. Besides, for extensive provisions, it is important to exploit the loop level parallelism and task level parallelisms [5]. On the other hand, not like as parallel compilers, existing frameworks that using task level parallelism interest programming exertion, which extends from needing to physically make and synchronize of tasks to needing to program in diverse dialects and ideal models. So because of this there is a requirement for frameworks that can exploit task level parallelism with the sequential type of programs.

## 1.2 Thesis Contribution

This thesis describes a method called Symbol-Table method which will exploits and identify task level parallelism at inner level of sequential C programs.The input to the normal gcc compiler is a sequential C program in which the procedures taken as functions(tasks). Than we generate a normal symbol table with specific command by gcc compiler in linux and an extended Symbol-table to get the additional info about the variables and functions as an data output.This method identify and detect the tasks to exploit the parallelism while maintaining the programs sequential steps. With this method we can identify the inner level dependencies between tasks and results in good performance improvements. This method is not much more applicable because of available compiler analysis, like by dependencies that occurs within the sequential C programs.

## 1.3 Project Overview

This method is mainly composed of two levels: a normal symbol table and an extended symbol table. The input given to the normal gcc compiler is a sequential C program in which the procedures taken as functions(tasks) with declarations. For generate a normal symbol table we provide a specific command to gcc com-

piler in linux. Then we use the information of that symbol table for generating the extended symbol table with additional information about variable's extended scopes, L/R-value attributes and inner level function dependency. This extended symbol table is generated by the use of previously generated normal symbol table on the basis of variable's scope, Declared line and referenced line of variables and functions with in the program. By that table we can identify the functions and variables those who are sharing the common variables and those who are accessing the different functions with extended scopes respectively. Then we can generate dependency graph based on that table's information to get the clear understandings about the dependencies which are exists in the program. Finally we can generate the Function graph for every variable in the program with the help of table's info and dependency graph's states.From that graph we can get the info about extended scoping of variables to identify and exploit the task level parallelism in the program. Then we can apply the parallelism with MPI or other parallel platforms to get optimized and error free parallelism.

## 1.4 Thesis Organization

The thesis is organized as follows: Chapter 2 discusses the Theoretical Background about parallelism. Chapter 3 describes the Literature review done for this thesis. Chapter 4 describes our proposed method design, implementation and results. This method is implemented on a simple C source program for exploiting the task level parallelism. Finally chapter 5 concludes with the summary of work done.

# Chapter 2

# Theoretical Background about parallelism

Types of Data dependence

Loop level Parallelism

Task level Parallelism

Summary

# Chapter 2

# Theoretical Background about parallelism

This chapter gives an idea about many types of data dependence, loop level and task level parallelism.

## 2.1 Types of Data dependence

This part describes about data dependence and its types. That data about dependence is covered in literature [7] [8] [9] and presented here.

Data dependencies are occurs when two or more iterations, statements or operations of a loop cycle can be executed in parallel.Basically four types of data-dependencies are there:

- 1. True dependency: (also known as Flow Dependency) it occurs between the two statements of a program, if the first statement write the data and the second other statement read it later.
  For example, in this program

  St1: c = a + b
  St2: d = c * 2

  So from these statements it is clearly shown that here is a true dependency exists between these statements St1 and St2, denoted as St1  St2, because St1 writes var c and St2 reads var c.

- 2. Anti Dependency: it occurs between the two statements in a program segment if the first statement reads the data and the second other statement writes it later.

  For example, in the program

  St1: c = a + b

  St2: a = d * 2

  So from these statements it is clearly shown that here is an anti dependency between St1 and St2, denoted by St1 $^{a}$ St2, caused by variable a.

- 3. Output Dependency: it occurs between the two statements in a program segment if the first statement writes the data and the second other statement again writes it later.

  For example, in the program

  St1: c = a + b

  St2: c = d * 2

  So from these statements it is clearly shown that here is an output dependency between St1 and St2, denoted by St1 $^{o}$ St2, because of variable c.

- 4. Input Dependency: it occurs between the two statements in a program segment if the first statement reads the data and the second other statement again reads it later.

  For example, in the program

  St1: c = a + b

  St2: d = a * 2

  So from these statements it is clearly shown that here is an input dependency between St1 and St2, denoted St1 $^{i}$ St2, caused by variable a.

So these types of dependencies can be overcomes by variable renaming technique [9]. The only actual dependency is true dependency.

Dependencies can be occur between parts or instances of statements in a loop cycle, when the same element(variable) of an array can accessed by two instances of statements. If these having dependencies related to the same loop cycle then the dependency is known as a loop independent dependency , and without synchronization they can be executed parallel in a concurrent way .
For example, in the program

for i = 0 to N-1
St1: p[i] = q[i] + r[i]
St2: s[i] = p[i] + 1
endfor

Here it is clearly shown that the dependency that is exist here among the instances of St1 and St2 are loop independent dependency; so, without synchronization they can be executed parallel in a concurrent way. In such type of cases, that loop is called as a parallel loop. however, if the instances relate to different-different loop cycles, then that type of dependency is called as a loop carried dependency, and without synchronization they can not be executed parallel in a concurrent way.
For example,

for i = 0 to N-1
St1: p[i] = q[i] + r[i]
St2: s[i] = p[i-1] + 1
endfor

So it is clearly shown that a loop carried true dependency is here between the St1 in iteration i and the St2 in iteration i + 1 where i = 0 to N - 1.

## 2.2 Loop level Parallelism

In light of the fact that this parallelism is obtained by having processors simultaneously operate the same type of operations over a same set of data thats by this type of parallelism is also called data parallelism

So here is a parallel loops example:

```
St1: ....
Loop1: for i = 1 to M
           x[i] = i
       end for
St2: ....
Loop2: for j = 1 to N
           y[j] = x[j]
       end for
```

This parallelism is normally actualized on shared memory frameworks utilizing a fork-join model.Above example shows that a program has two parallel type loops. This source might execute like it takes after. St1 will be executed by solitary thread called master thread. When Loop1 is arrived at, M-1 worker threads are made, and every freely executes a cycle of Loop1 together with the master thread. At the end of the Loop1 and before it returns to execute St2 the master thread holds up the things till the worker thread finish their operations, . Likely, when the Loop2 is reached, N-1 worker threads are made to execute the cycles of the loop in the parallel way. Figure 2.1 illustrate the execution process of the above program with an graph, where nodes shows the threads execution and edges shows the threads dependencies.

In most of the parallel processors, a thread is to be assigned by more than one iteration typically, since one emphasis for every thread may prompt inadmissible overhead. The task

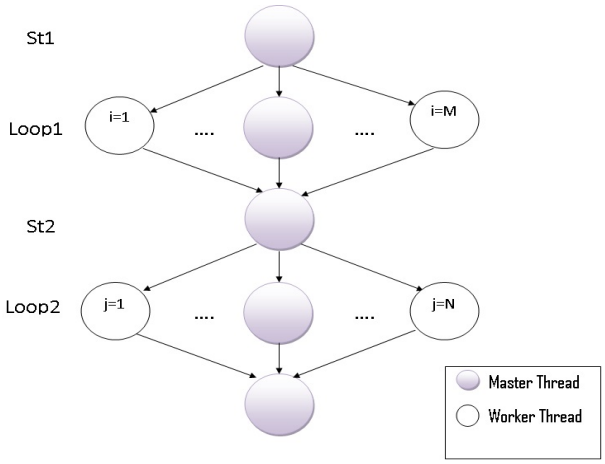


Figure 2.1: Execution graph for above parallel loops program

of emphases to threads is defined as scheduling the loop cycles. This scheduling may be carried out statically or dynamically [10] [3].

The Loop level parallelism's main point of focus is parallelizing compiler research [1] [2] [3], After all, it is not without the limitations [4].

1. If a loop is able to be parallel then a parallelizing compiler system must be able to determine statically. In some of the cases, the way in which the loop is coded, like that,suppose dependency exists when it is not able to prove rather then; otherwise such type of dependency may not be occur.

2. the appearance of these procedures in a loop of program causes the analysis to be stable. Therefore, in the modular programs the parallelism may not be totally exploited.

3. Data-parallelism may be useful when the data size is fit to the parallel machine's size. A literature [5] shows that the physical demands of the issue frequently make it difficult to subjectively build the data set's size for some of applications, and the parallel machine's resources doesn't completely use by data parallelism alone. Additionally, synchronization and communication overhead is developed with machine size with data parallelism; to make it less beneficial, apply more no of processors to a single data parallel computation [11].

## 2.3 Task level Parallelism

It is an another type of parallelism which is a group of co-operating tasks. A unit of computation as task or procedure which can be as coarse-grain as a procedure invocation or as fine-grain as an arithmetic operation that executes more and more number of instructions. When the independent tasks are executing concurrently then This type of Parallelism occurs. Tasks which are executing concurrently are not in limited range to operating same type of set of operations in comparison to data-parallelism.

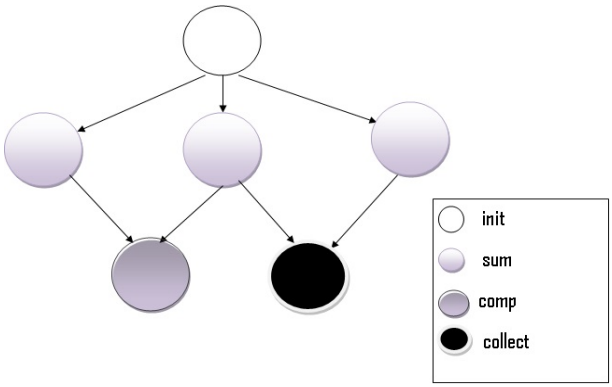


Figure 2.2: An example type of task graph

The thought of data dependence could be reached out to tasks. In frameworks where tasks are indicated by the data stream stipulations around the tasks, when a task P generates some data worth which is needed by another task Q then it is said that task Q is dependent on task P. Subsequently, task Q can't begin its execution until task P has finished its execution about the data. In frameworks where the tasks are well synchronized to concurring a particular order, for example, the successive order of data-access of the tasks, at that point when task P goes in execution before task Q in execution then it is also said that task Q is dependent on task P, furthermore the same data is written by either P or Q. For that situation, task Q can't begin its execution until the task P has completed its execution of the data.In both of the situations, we are referring the task P as the pre-requisite task and task Q referred as the dependent type task on P. A graph i.e. which is used to represent the dependencies among the tasks is called task graph. The task graph

is a directed type acyclic graph in which the nodes are representing the tasks and the edges are showing the dependencies between the tasks. The source and sink of a dependency edge is a pre-requisite task and a dependent task respectively. In general, when a pre-requisite task completes its execution then only a dependent task can execute.

For example, A task graph is represented in figure 2.2 of a program. In this the task is taken as a procedure invocation. The program execution would be follows like as. First of all Task 'init' will execute because of no pre-requisite tasks it has. After the completion of the task 'init' three 'comp' type tasks will be execute in parallel. Another task 'sum' will begin its execution when the first two 'comp' tasks will finish their executions, and the another task 'collect' will executes when the last two 'comp' tasks will finish their execution. So it is clearly shown in example that the results of parallelism occurs from executing the 3 'comp' tasks in parallel, and the tasks 'sum' and 'collect' are executing concurrently.

In general the Task-level parallelism is much more flexible than the Loop-level parallelism. Because as example, the tasks can be executed concurrently which are performing the same set of operations across the same set of data, compare to as data parallelism. Furthermore,the tasks can be executed concurrently which are performing different type of operations on the different type of sets of data also.

Nonetheless, the adaptable behavior of this task-level parallelism prompts a few drawbacks. As opposed to building parallelizing tools concentrating on a particular programming developed loop, different languages or frameworks have been made to help diverse kinds of task-level parallelism. According to their meaning of the task the frameworks changes; according to the time on which tasks are made;according to the systems which is used to help in task synchronization and communication between tasks; and additionally according to the programming languages and standards used to exploit the parallelism. Hence, to exploit the task-level parallelism, a developer should first pick a framework, and either

broadly adjust the sequential type of C-programs or modify them in the standard or programming languages needed by the framework.

## 2.4 Summary

This chapter gave a conceptual background on data dependencies, loop level and task level parallelism. It likewise discussed about a few points of interest or advantages of task level over loop level parallelism.

# Chapter 3

# Literature Review

Review of Related Work

SCHEDULE

TDFL and LGDF2

COOL

Jade

pTask

Other Systems and Languages

Summary

# Chapter 3

# Literature Review

The chapter represents and discusses about the related work for exploitation of task level parallelism. This chapter depicts a few frameworks that supports task-level parallelism, centering on the system for expressing parallelism and the programming exertion. Specifically, it depicts SCHEDULE, LGDF2, TDFL, COOL, and Jade. Different frameworks, for example, Hypertool, Fortran M, PYRROS, and Fx are likewise talked about.

## 3.1 Review of Related Work

### 3.1.1 SCHEDULE

SCHEDULE [12] is a Fortran language type library package which is used to express inter-task dependencies. A task as a subroutine call is taken in SCHEDULE system. The developer supply a remarkable identifier for each one task to unequivocally determines its dependence connections, the amount of essential tasks, and the amount of dependent tasks and identifiers those are related to these tasks. In actuality, the developer manually produces task graphs.

SCHEDULE system has some advantages like, During the process of task synchronization it has little run time overhead because dependencies between the tasks are specified explicitly by the programmer. However, The SCHEDULE has some disadvantages like.

1. From a sequential c program or complex algorithm it is not possible always to deduce it to correct task graph but here the programmer must have to describe

the graph in the form of an algorithm for create a task graph.

2. Mainly in the time of the maintenance the system programmer must have to specify the existing dependencies between the tasks manually,that is may be erroneous process.

3. The task graphs which are used in SCHEDULE system are static in nature thats by the dependencies can't be changed among the tasks and once the program begins to execute the new tasks cannot be merged to the task graph. So the programmer must have a priori knowledge about the structure of computation to generate the task graph.

### 3.1.2 TDFL and LGDF2

TDFL stands for Task Level Dataflow Language [13] and LGDF2 stands for Large Grain Data Flow [14] both are the parallel languages that provide the facilities to programmers to express the synchronization and concurrency with data-flow graphs. In data-flow graph the node shows task and edge represent the data-flow between the nodes. In both of languages the task is taken as a procedure. In a given data-flow graph, the framework executes tasks as per the data-flow obligations.

TDFL and LGDF2 both are the same type of languages and have some advantages over the SCHEDULE system.

1. They require only specification of data-flow constraints among the tasks, in place of the explicit specified inter task dependencies.

2. They propose graphical interface to understand the specification about data-flow graphs.

3. They propose the programming platforms to support the dynamic task creation. After all, these systems still trouble the developer with the throwing of algorithms and programs into data-flow diagrams.

### 3.1.3 COOL

It is a Concurrent Object-Oriented Language (COOL) [15] which is determined by C++ and using some constructs to point out the concurrency. In COOL a task

is taken also as a procedure that is proclaimed as a parallel task. An instance of task is dynamically made at the point of parallel procedure. COOL executes the tasks consequently that work on distinctive objects in the parallel way. COOL uses mutex procedure.

For further exploiting the concurrency inside an object, developer can make the object from littler objects.

COOL has many advantages compare to others.
it gives to the programmer more commonplace programming ideal model. Furthermore,it gives mutex for the synchronization and creation of the tasks. Moreover, there is no compelling reason to define the algorithms as task or data-flow graphs.

COOL also has two drawbacks with respect to physically making objects from littler objects.
1. Created object's accesses could be prohibitive. Such as, a row wise parallel type matrix in a program of matrix class can't be simultaneously accessed as column wise.
2. If the object is a vector of little base objects,on the other hand, requests comparative effort to parallel programming utilizing explicit concurrent primitives.

### 3.1.4 jade

Jade [18] [17]is also a parallel type programming language which is based on C++. It permits the programmer to specify the dynamic coarse-grain parallelism. In Jade a task is taken as a program block that is commented by programmer with data utilization information of that block(side-effects). To define a task as a program block, the programmer constructs the data being accessed in the block; such code is referred to as side-effect. The Jade uses Tokens to point out the side-effects of shared data.

Jade likewise goes to exploit concurrency around objects and inside an object. Dissimilar to COOL, in which programmer should physically form the objects, a Jade programmer just need to utilize tokens to logically divide the objects.The major disadvantage is number of tokens may be large for larger data. Such as, to exploiting the parallelism around tasks accessing to disjoint different different

parts of a vector.

### 3.1.5 pTask

pTask [16] is a compiler which take a sequential program as input and provide a parallel program as output, exploiting the task level parallelism with the help of SIGMA II tool kit. This illustrates the functions of the compile-time module and run-time module.

Disadvantage is dependency analysis is taking place at run time module so it has High run-time overhead.

### 3.1.6 Other Systems and Languages

Pyrros [20] implemented by Yang and Gerasoulis , it allows only static task-parallelism and estimates of task execution times must be provided by the programmers. As a output the system produces the program and scheduling of tasks to the processors.

Fortran M [11] is a language which is subset of Fortran extensions. A programmer uses its features to define tasks which are like procedures, to explain that the tasks are to be executed in a concurrent way, and to define communication around tasks. Tasks communicate with the help of sending and receiving information individually, i.e., by message passing task synchronization is achieved.

Gross, OHallaron, and Subholk Fx [21] is a compiler which is designed and implemented based on HPF. To exploit the task parallelism it allows the programmers to extends HPF. In which task is defined as a procedure invocation, but in parallel sections of program which are defined by programmer it prevent the parallelism. So, programmers require to supply directives that can specify the input and output parameters of each and every task, and the mapping of tasks onto processors.

## 3.2 Summary

This chapter gave a review of related work, with the existing frameworks or systems, shows more programming efforts are required to gain task level parallelism.

# Chapter 4

# Proposed method and implementation

**Introduction**

**Our Proposal**

**Results**

**Summary**

# Chapter 4

# Proposed method and implementation

## 4.1 Introducton

The proposed method of this research work motivated by the task level parallelism [16] to exploit the parallelism at inner level of functional dependencies in a sequential c program. The idea of the proposed method developed here only to identify the inner level dependencies of tasks(functions) statically at compile time. For exploiting the concurrency with Task level parallelism we have implemented a simple sequential c program with four functions on the Linux OS with gcc compiler. Here we are targeting the general workings of the compiler about symbol table creation.From that symbol table we are generating the Extended symbol table with additional information about variables and functions of program. From that table we can identify the inner dependencies of functions in a program and by the functional graph we can show the relation between functions about the dependencies. We are using "NetBeans" tool with java platform for generating the dependency graph of program by using that Extended symbol table data in java program.

## 4.2 Our Proposal

**Exploitation of Task level parallelism with Symbol-Table method:**
our objective is "if 2 functions have common variable then they can not execute

parallel(because they are accessing the same memory location of common variable). How these functions are accessing the same memory location?" For proving this we are using symbol table analysis.

According to our proposal there are three levels to solve that problem.

- Generalized Symbol-table.
- Extended Symbol-table.
- Dependency and Function graph representation.

first we are taking a simple sequential c program as input to the gcc compiler with the Linux OS.Then with the specific commands
(i.e. gcc filename.c -o filename
gcc -c filename.c
readelf -a filename.o)
We can generate internal process of execution and compilation of program(Executable Locate File(ELF)). It has all information about program like ELF Headers, section headers, section groups, program headers, key to flags, relocation sections, unwind section and most important "Symbol-table". From that we can generate the generalized Symbol-table with information like:
1. Name of Variables and Functions
2. Characteristics Class
3. Token id
4. Scope of Variable and Function
5. Declared line
6. Referenced line
7. Other information like parameters used by functions.

Then for getting the Scope information we generate the Symbol-table for every function which is present in the program.

After generating the generalized Symbol-table with all this information we can

generate the second level of this method which is called "Extended Symbol-table" with the additional information about extended scopes of variables and the L-value/R-value attributes of the variables as output with the help of previously generated generalized Symbol-table. In that Symbol-table clear information is provided about extended scopes of the variables and L-value/R-value attributes of the variables. For "Extended Symbol-table" the attributes are:

1. Name of Variables and Functions.
2. Scope of Variables and Functions.
3. Declared Line of Variables and Functions in program.
4. Referenced Line of Variables and Functions.
5. Extended Scope of Variables and Functions.
6. L/R value attributes.

On the basis of Declared line and Referenced line we can draw the dependency graph with the help of java based program of NetBeans tool. And on the basis of Extended scope and L-value attribute we can represent the function graphs manually regarding the information about variables those are changing their scopes with L-value attribute. From these graphs we can easily identify the dependencies among functions and which functions can execute parallel to each other.

### 4.2.1 Basic concepts behind the proposed method

This section will give you the little bit description about working principles of Symbol-table and scopes. And also little bit about L-value/R-value concept, Binding and function graph.

#### Symbol-table and scopes

Basically Symbol-table is a type of data structure which is used by compilers to store the information about the source program. In figure 4.1 the structure of a compiler is presented. Basically compiler has two parts:

- 1. Analysis part: (also known as Front end) it consist of Lexical analyzer, Syntax analyzer and Semantic analyzer. The Analysis part collects all in-

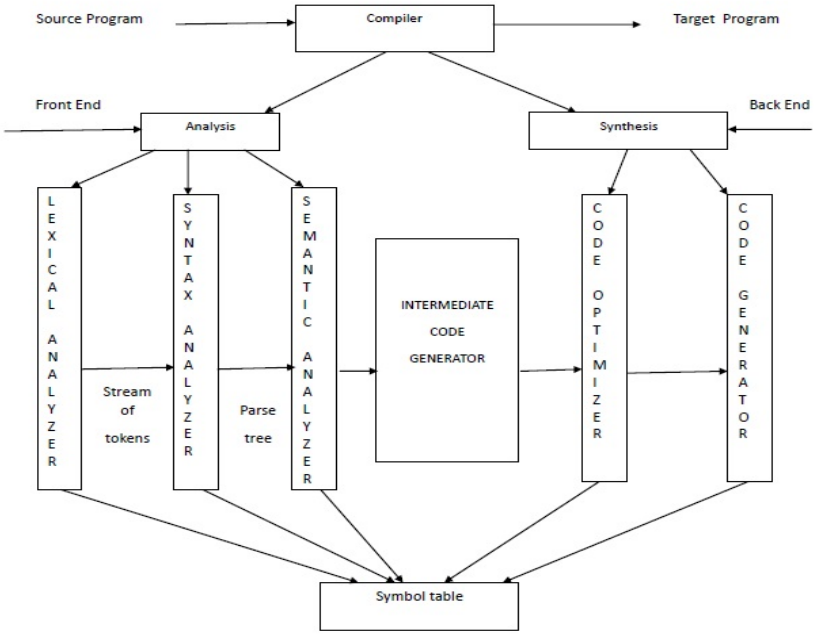


Figure 4.1: The structure of a Compiler.

formation about the source program and store it in a data structure that is called Symbol-table. Which is passed to the Synthesis part.

- 1. Synthesis part: (also known as Back end) it consist of Code optimizer and Code generator phases. The Symbol-table is used by Synthesis part.

A compiler record the variable names and collect the information regarding various attributes of every name which are used in source program. These attributes provide the information about the its scope, its type, procedure names, number of its arguments and their types,storage allocated for a name, method of passing of each argument and the return type.

Symbol-table is created by the compiler from AST just before the byte code is created. The Symbol-table is very much responsible for calculating the scopes of each identifier in the source code. Symbol-table typically required to support more than one declaration of the same identifier in a source program. The portion of a program to which the declaration applies is called the declaration of a scope. We shall implement the scopes by generating the separate Symbol-table for each

and every scope.

In general The names are binded to their associated information then it is called Symbol-table.

**Binding:-** Ability to provide name objects such as variables, functions and their types in the programming languages is an important concept. Each such type of named objects will have a declaration, where the name is defined as a synonym for the object, that is called binding. The declaration of a name has limited scope.

In general a pair of a name and its associated information is called binding and a Symbol-table has a list of such type of binding.For accomplish the Symbol-table we need to perform a number of operations on Symbol-table. That are:

- **1. Empty**
- **2. Binding**
- **3. lookup**
- **4. Enter**
- **5. Exit**

**L-value and R-value expressions**

The expressions that refers to memory locations are known as L-value expressions.An l-value represents a store region called "locater" value, or a left value, showing that it can be appear only on the left side of equal sign(=). L-values are often as identifiers. L-value expressions that are referring to modifiable locations are called "modifiable l-values".For example:

A = 3;

Here, A represent an identifier and name which denotes the storage location. and 3 is denotes what stored at the location.
Any of the following C expressions can be L-value expressions:

- **1.** An identifier of integer,float,pointer,structure or union type.
- **2.** A subscript ([]) expression that doesn't evaluate to an array.
- **3.** A member selection expression ($-->$ *or.*).
- **4.** A unary indirection (*) expressionthat doesn't refer to an array.
- **5.** An L-value expression which is in parentheses.
- **6.** A constant object like as a non-modifiable L-value.

The term R-value is used to describe the actual value of an expression and to differ it from L-value.
An assumption is there that is: "All L-values are R-values but vise-versa is not right".
For example:

```
int var;
var = 3;
```

Here it is correct representation but

```
3 = var;
(var+1) = 3;
```

it is not a correct representation because assigning to them makes no semantic means i.e. there's no where to assign to.
**L-value to R-value conversion is possible**
For example:

```
int p = 5;
int q = 6;
int r = p + q;
```

Here, p and q are L-values in first two lines and in third line these are R-values.
**But R-value to L-value conversion is not possible** because this would violate the nature of a L-value.

### Function graph

It is a graph which shows the relationships among the functions like as one to one, one to many, many to many or self loops. It shows the clear view about dependencies between functions through the scoping(actual or temporary).

On the basis of **Dependency graph and Extended Symbol-table's information** we can generate the Function graph. In figure 3.2 a Function graph is shown.

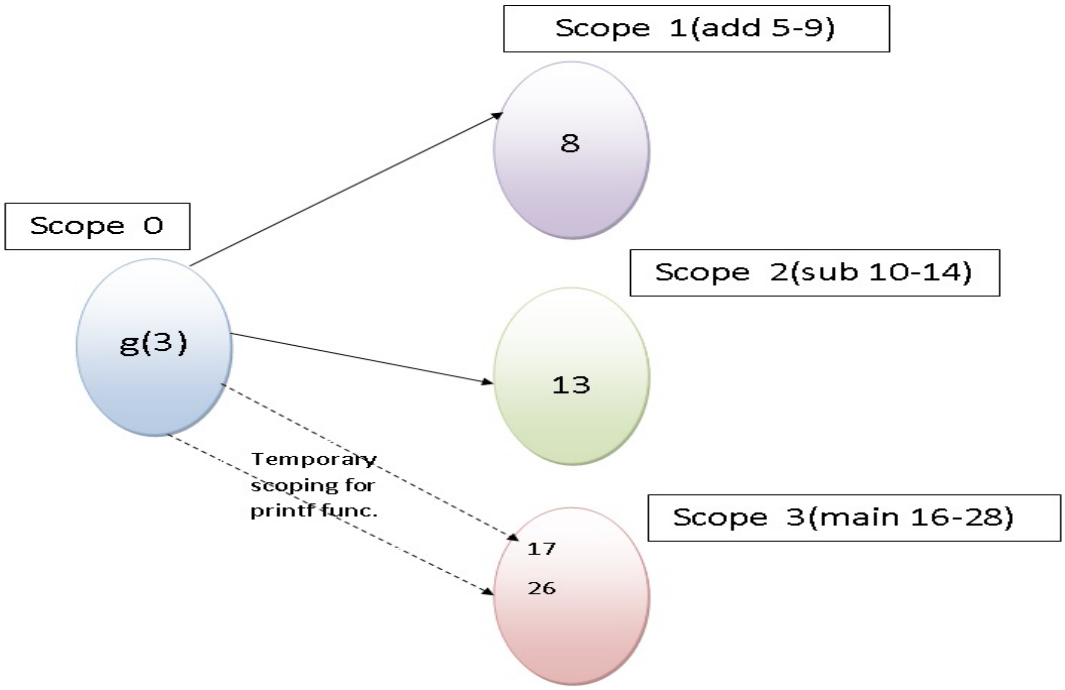


Figure 4.2: A sample of Function graph.

Here, the circles are representing the different-different scopes of the source program which is indirectly representing the different functions. The numbers in the circles representing the line numbers of the source program. The solid line representing the Actual scoping between functions and the dotted line representing the Temporary scoping i.e. for printf functions or for R-value expressions.

So, from the dependency graph and this Function graph we can evaluate the results of our proposed method.

### 4.2.2 Design and implementation of the proposed method

Our proposed method is based on Symbol-table design and implementation. Here Symbol-table designing process is divided in to mainly two parts.

- Generalized Symbol-table.
- Extended Symbol-table.

So for designing and implementing using this Symbol-table analysis method we have taken a simple and small sequential C-program. The source code of that program is given here:

**Program source code**

```
#include <stdio.h>
#include<conio.h>
int g = 10;
 int I = 15;
Int *h;
int add(int a, int b)
{
        a = a + b;
        g = a + b;
}
int sub(int a, int b)
{
        a = g - b;
        g = *h - b;
}
int mul(int *a, int *b)
{
        *a = *a * *b;
        *b = *h * *a;
}
int div(int *a, int *b)
{
        *a = *a / *b;
        *h = *h / *b;
}
int main ()
{
printf(in main1 I m = %d %u, g,&g);
Int x = 10;
int y = 5;
 int sum = 0, minus = 0;
```

```
int multi = 0, divide = 0;
 h = &I;
sum = add(x,y);
minus = sub(x,y);
 multi = mul(&x , &y);
 divide = div(&x , &y);
printf(in main2 I m = %d %u %d %d %u,  g,&g,x,I,&i);
return 0;
}
```

it is a 40 line simple source program which has mainly four functions i.e. add(), Sub(), mul(), div(). These functions are sharing some common variables.The variables are used in this program are g,h,i,a,b,x and y in which g,h and i are global variables and rest are local variables of different functions. Because of the data inconsistency these functions can't execute with parallel platform. So, For finding the inner dependencies among the functions we are using this Symbol-table method.

So first we generate the Symbol-table by gcc compiler with specific commands on Linux platform. Than from that table we can generate the "Generalized Symbol-table" using unordered-listed data structure on the basis of some important attributes like Name, Char class, Token id, Scope, Declared line, Referenced Line and other info about variables and functions in the program.

We can generate a separate Symbol-table for each scope(function) which is used in the source program.The **"Generalized Symbol-table"** for that above program is shown in table 4.1, 4.2, 4.3, 4.4, 4.5:

Table 4.1: **Generalized Symbol-table with Main Function**

| **Name** | **Char class** | **Token id** | **Scope** | **Dec. line** | **Ref line** | **others** |
|---|---|---|---|---|---|---|
| g | var int | $id < 1 >$ | o(global) | 3 | 9,13,14,28,38 | - |
| i | var int | $id < 2 >$ | o(global) | 4 | 5,14,19,24,38 | - |
| h | ptr var int | $id < 3 >$ | o(global) | 5 | 14,19,24 | - |
| add | func. int | - | 1 | 6-10 | 34 | two parameters a,b |
| sub | func. int | - | 2 | 11-15 | 35 | two parameters a,b |
| mul | func. int | - | 3 | 16-20 | 36 | two parameters a,b |
| div | func. int | - | 4 | 21-25 | 37 | two parameters a,b |
| main | func. int | - | 5 | 26-40 | - | - |
| x | var int | $id < 12 >$ | 5 | 29 | 34,35,36,37,38 | - |
| y | var int | $id < 13 >$ | 5 | 30 | 34,35,36,37 | - |
| sum | var int | $id < 14 >$ | 5 | 31 | 34 | - |
| minus | var int | $id < 15 >$ | 5 | 31 | 35 | - |
| multi | var int | $id < 16 >$ | 5 | 32 | 36 | - |
| div | var int | $id < 17 >$ | 5 | 32 | 37 | - |

Table 4.2: Symbol-table for **add() Function**

| **Name** | **Char class** | **Token id** | **Scope** | **Dec. line** | **Ref line** | **others** |
|---|---|---|---|---|---|---|
| a | var int | $id < 4 >$ | 1(local) | 6 | 8,9 | - |
| b | var int | $id < 5 >$ | 1(local) | 6 | 8,9 | - |

Table 4.3: Symbol-table for **sub() Function**

| **Name** | **Char class** | **Token id** | **Scope** | **Dec. line** | **Ref line** | **others** |
|---|---|---|---|---|---|---|
| a | var int | $id < 6 >$ | 2(local) | 11 | 13 | - |
| b | var int | $id < 7 >$ | 2(local) | 11 | 13,14 | - |

Table 4.4: Symbol-table for **mul() Function**

| **Name** | **Char class** | **Token id** | **Scope** | **Dec. line** | **Ref line** | **others** |
|---|---|---|---|---|---|---|
| a | var int | $id < 8 >$ | 3(local) | 16 | 18,19 | - |
| b | var int | $id < 9 >$ | 3(local) | 16 | 18,19 | - |

Table 4.5: Symbol-table for **div() Function**

| **Name** | **Char class** | **Token id** | **Scope** | **Dec. line** | **Ref line** | **others** |
|---|---|---|---|---|---|---|
| a | var int | $id < 10 >$ | 4(local) | 21 | 23 | - |
| b | var int | $id < 11 >$ | 4(local) | 21 | 23,24 | - |

This is a Generalized Symbol-table for above program which contains the information about variables and functions like name,type,token-id,scope,declared line in program, referenced line in program etc. Now on the basis of scope,declared line and referenced line we can generate the "Extended Symbol-table" with additional information about variables like extended scopes and L/R-values attributes.
The Extended Symbol-table over the Generalized Symbol-table is shown in table 4.6. From this table we can get the additional info about variables like Extended scopes and L/R-value on the basis of reference lines. We can get the common variables from this table those are shared by functions.The variables those have Extended scopes with L-value attributes are comes in focus only.

So with the help of this additional information we can go to generate for inner dependency graph and Function graphs as result.

Table 4.6: **Extended Symbol-table for whole program**

| **Name** | **Scope** | **Dec. line** | **Ref. line** | **Extended scope** | **Value(L-R)** |
|---|---|---|---|---|---|
| g | 0(global) | 3 | 9 | 1 | L-value |
| g | 0 | 3 | 13 | 2 | R-value |
| g | 0 | 3 | 14 | 2 | L-value |
| g | 0 | 3 | 28 | 5 | - |
| g | 0 | 3 | 38 | 5 | - |
| i | 0(global) | 4 | 5 | local | - |
| i | 0 | 4 | 14 | 2 | R-value |
| i | 0 | 4 | 19 | 3 | R-value |
| i | 0 | 4 | 24 | 4 | L-value |
| i | 0 | 4 | 38 | 5 | - |
| h | 0(global) | 5 | 14 | 2 | R-value |
| h | 0 | 5 | 19 | 3 | R-value |
| h | 0 | 5 | 24 | 4 | L-value |
| add | 1 | 6-10 | 34 | 5 | - |
| a | 1(local) | 6 | 8,9 | local | L-val,R-val |
| b | 1(local) | 6 | 8,9 | local | R-val,R-val |
| sub | 2 | 11-15 | 35 | 5 | - |
| a | 2(local) | 11 | 13 | local | L-value |
| b | 2(local) | 11 | 13,14 | local | R-val,R-val |
| mul | 3 | 16-20 | 36 | 5 | - |
| a | 3(local) | 16 | 18,19 | local | L-val,R-val |
| b | 3(local) | 16 | 18,19 | local | R-val,L-val |
| div | 4 | 21-25 | 37 | 5 | - |
| a | 4(local) | 21 | 23 | local | L-value |
| b | 4(local) | 21 | 23,24 | local | R-val,R-val |
| main | 5 | 26-40 | - | local | - |
| x | 5 | 29 | 34 | local | - |
| x | 5 | 29 | 35 | local | - |
| x | 5 | 29 | 36 | 3 | L-val,R-val |
| x | 5 | 29 | 37 | 4 | L-value |
| x | 5 | 29 | 38 | local | - |
| y | 5 | 30 | 34 | local | - |
| y | 5 | 30 | 35 | local | - |
| y | 5 | 30 | 36 | 3 | R-val,L-val |
| y | 5 | 30 | 37 | 4 | R-val,R-val |
| sum | 5 | 31 | 34 | local | - |
| minus | 5 | 31 | 35 | local | - |
| multi | 5 | 32 | 36 | local | - |
| divide | 5 | 32 | 37 | local | - |

## 4.3 Results

As we have completed both parts of this Symbol-table analysis method.Now we can generate the inner-Dependency graph with the help of both tables with Declared line and Reference line. The figure 4.3 shows the dependency graph on the basis of declared and referenced line. This dependency graph is generated by the NetBeans tool which is a java based platform tool. With the help of a java program we are creating a .txt file. Than we are putting the information about declared line, referenced line and number of lines from the generated tables into the .txt file. Than we can execute that java program to get the inner-dependency graph. In this graph the nodes or circles represent the line number at which the variable declared or referenced in the program. And the edges represent the data dependencies from declared line node to referenced line node.

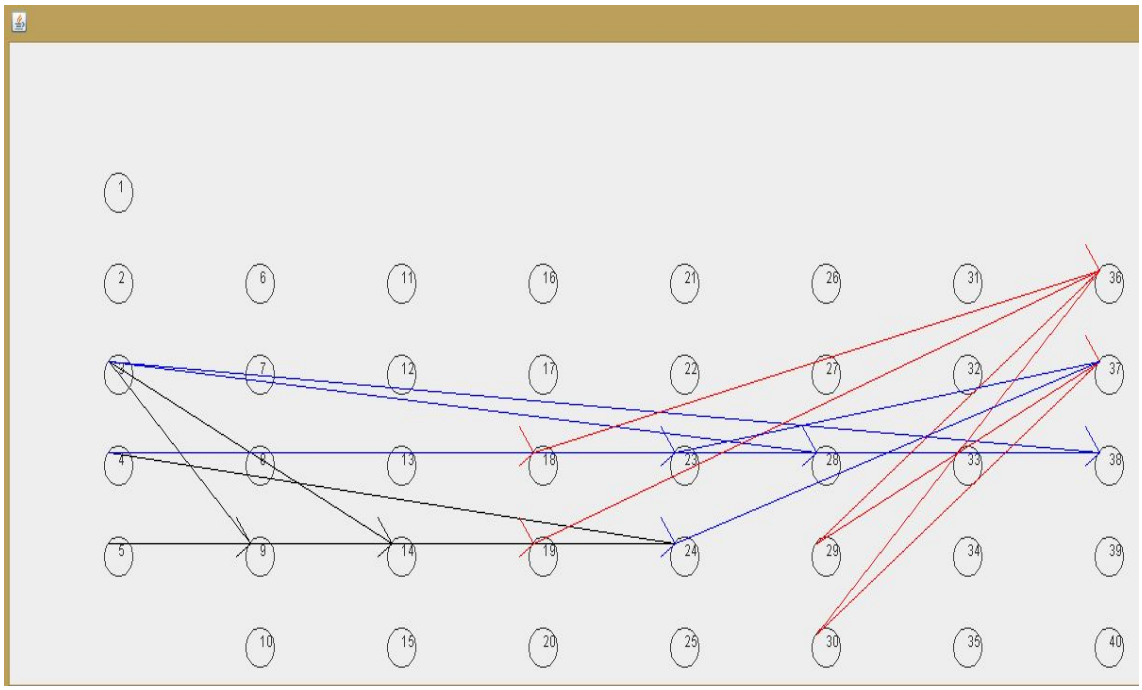


Figure 4.3: Dependency graph based on Table's info

In this Dependency graph, black color edges representing Data dependencies between nodes(lines) and blue color edges representing temporary dependencies and red color edges representing the referential dependencies. With the help of

this dependency graph and Extended Symbol-table's additional information we can generate the "Function graph" finally.

**Function graph**

On the basis of Dependency graph and Extended Symbol-table's information like L-value attribute(only) and extended scope of variables through referenced line we can get the data to draw the Function graph.

The data is like that:

- g{0,3} $\longrightarrow$ [{1,9}{2,14}{5,(28,38)}]
- i{0,4} $\longrightarrow$ [{4,24}{5,38}]
- h{0,s} $\longrightarrow$ [{4,24}]
- x{5,29} $\longrightarrow$ [{3,36,18}{4,37,23}]
- x{5,30} $\longrightarrow$ [{3,36,19}{4,37,(23,24)}]

The expressions is written as like: the description of first expression data is: Here, the left hand side data of an arrow represents the info about variable, for which we have to generate the Function graph.The info is, g is a variable which is defined in scope 0 and declared at line 3 in the program.And the right hand side data of an arrow represents the info about variable g that what are the extended scopes of g with-in it is used.i.e. in scope 1 on line 9 (which is referenced line of g) variable g is used.like that it is all the data about var g.

Like that for variables h,i,x and y that data is given by expressions. on the basis of that data we can generate the graph. From these graphs we can conclude some assumptions about explot the task level parallelism that:

- **1.** 1 to 1 scoping is allowed between scopes.
- **2.** self-loop is allowed(local scoping).

- **3.** 1 to many scoping is not allowed.

Figure 4.4 shows the Function graph for var g. The Function graph clearly shows

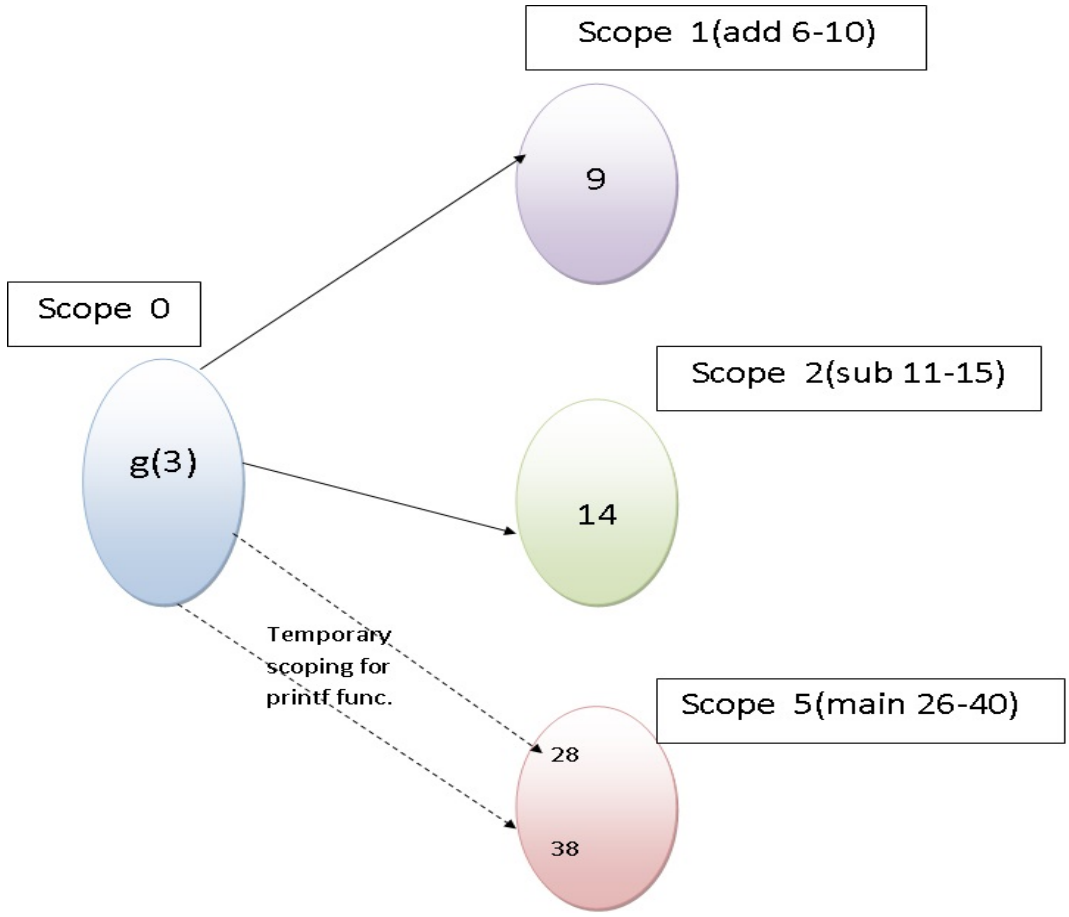


Figure 4.4: Function graph of g var.

that var g is declared in scope 0 at line 3 which is global scope and referenced to different-different extended scopes, in which in scope 1 which is add() function, at line number 9 with L-value attribute it is used.Like that in scope 2 it is used also.In scope 5 it is also used but not with L/R-value attributes, so it is showing as temporary scoping for printf() function.

By analyzing this particular graph we can say that there are problem for going to parallel with these referencing scopes (functions). Because both functions are using the same var g. In these scopes the value of var g can be modified, so they can't go parallel to each other.

Like that Figure 4.5 shows Function graph for var i, Figure 4.6 shows Function graph for var h, Figure 4.7 shows Function graph for var x and Figure 4.8 shows Function graph for var y.

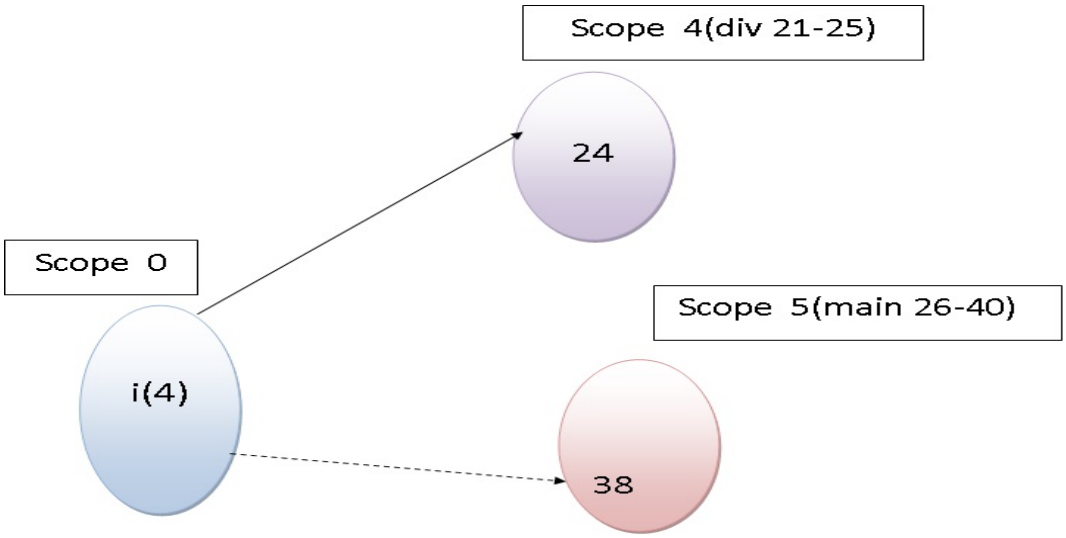


Figure 4.5: Function graph of var i.

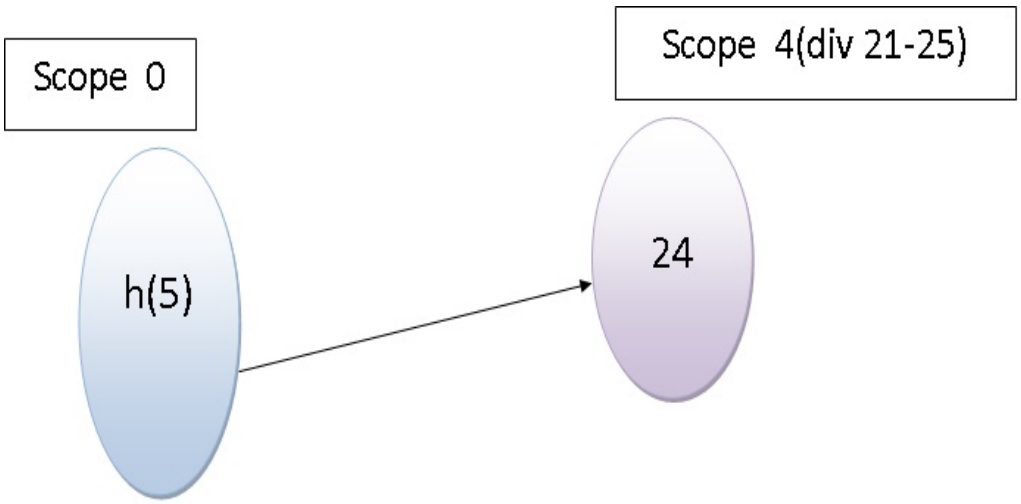


Figure 4.6: Function graph of var h.

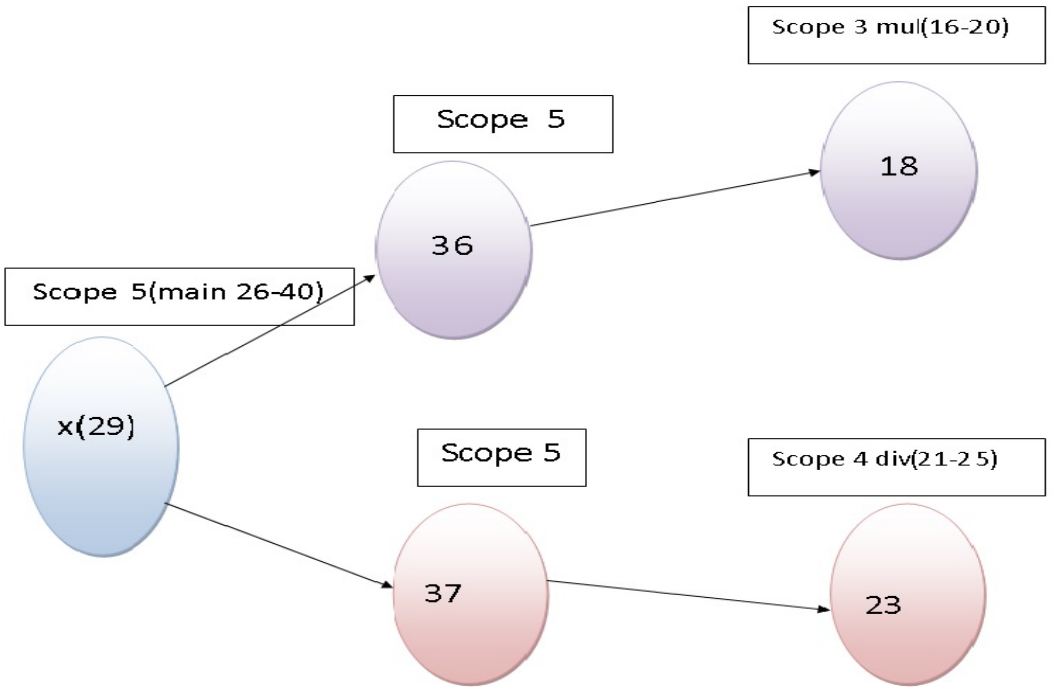


Figure 4.7: Function graph of var x.

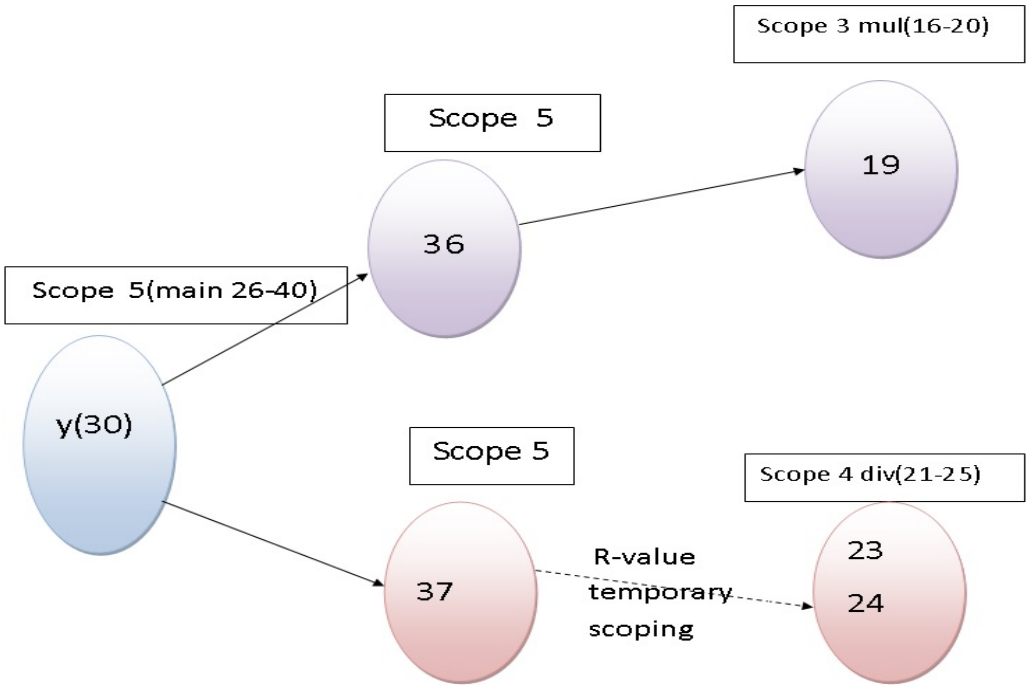


Figure 4.8: Function graph of var y.

From figure 4.5 which is showing Function graph for var i, we can evaluate that var i is used by only scope 4 (div() function) with actual scoping(Extended scope + L-value attribute only). By scope 5 it is used as temporary scoping.So

there are no problem with var i. Like wise figure 4.6 showing function graph for var h, in which it is clearly shown that var h has only one actual scoping in scope 4. According to assumption 1 to 1 scoping is allowed between scopes so there are no problem with var h.

Figure 4.7 showing the Function graph for var x, in which because of referencing dependency which is identified by dependency graph var x is referenced through scope 5 to scope 3. by graph it is clear that there 1 to many actual scoping is present between scopes for var x. So these specific scoping-functions are sharing common var x, thats by they can't execute parallel to each other. Like wise figure 4.8 showing function graph for var y, in which it is clearly shown that var y has 1 to 1 actual scoping in scope 3. So there are no problem with variable y.

After analysis of all these graphs finally we can say that:

- add() function and sub() function can't be execute parallel because of var g's 1 to many actual scoping.
- mul() function and div() function also can't be execute parallel because of var x's 1 to many referencing actual scoping.
- add() and mul() or add() and div() may be execute parallel because there are no dependency or scoping conflict between these functions.
- Also sub() and mul() or sub() and div() may be execute parallel because there are no dependency or scoping conflict between these functions.

## 4.4 Summary

This chapter describes the design,implementation and results of our proposed method i.e. Symbol-table method. We have implemented this method on a simple C-program with four simple functions. From the parts of this method we are getting all the information about variables and functions and finally we are getting the results in form of Dependency graph and Function graph. From these graphs we can exploit the parallelism as a Task Level Parallelism in the program.

# Chapter 5

# Conclusion and Future work

**Conclusion**

**Limitations and Future Work**

# Chapter 5

# Conclusion and Future work

## 5.1 Conclusion

The Loop level parallelism had some limitations which are solved by the task level parallelism. After all, with the currently existing frameworks also those support this type of TLP, a system programmer must make the executable updates to the sequential C source program to achieve the required level of task parallelism.In this thesis this work has been done using a method is called Symbol-table method at the time of compilation. This method has basically two different parts i.e. Generalized Symbol-table generation and Extended Symbol-table generation.By these parts of the method we can get the additional information like reference line, declared line, scope, extended scope and L/R-value attributes about the variables and functions which is used in the source program. With the help of this information first we can draw the program dependency graph and after that with whole information about variables we can generate the Function graph for each variable.From that graph we can clearly evaluate the inner level dependencies among the functions and extended scoping information about variables. On the basis of that information we can detect and exploit the task level parallelism.Then we can apply the parallelism with MPI or other parallel platforms to get optimized and error free parallelism.

## 5.2 Limitations and Future Work

This method has some limitations because of the quality of available compiler's analysis, data dependencies that occurs in the sequential C-programs and the choice of only functions as tasks as units of parallelism.The Symbol table method we are using to get information about variables and functions, is generated manually. So the correct exploitation of parallelism depends on the correctness of the symbol table, which totally depends on the programmer. So we suggest some future work for Symbol-table method to exploit the Task Level parallelism in general. To generate the generalized and extended symbol tables with correct and all additional information for program we have to improve the quality of compiler's analysis. The next one is that if we want to implement an analysis module that provides side-effects of the statements in a program. Then, in place of providing a wide set of features like as side-effect analysis,code generation and manipulation and dependency analysis, this will concentrate on the information that would help systems to support task level parallelism. This module should be able to accurately describe the accesses of Symbol-table to dynamic data structures such as linked-lists and trees.To identify the recursive procedure's data accesses is also a challenge.

# Bibliography

[1] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren, "The parascope parallel programming environment," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 244–263, 1993.

[2] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiler optimizations for fortran d on mimd distributed-memory machines," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 86–100, ACM, 1991.

[3] C. D. Polychronopoulos, M. B. Gikar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten, "The structure of parafrase-2: An advanced parallelizing compiler for c and fortran," in *Selected papers of the second workshop on Languages and compilers for parallel computing*, pp. 423–453, Pitman Publishing, 1990.

[4] R. Eigenmann and W. Blume, "An effectiveness study of parallelizing compiler," in *Proceedings 20th International Conference Parallel Processing 1991*, vol. 2, p. 17, CRC Press, 1991.

[5] J. Subhlok, J. M. Stichnoth, D. R. O'hallaron, and T. Gross, "Exploiting task and data parallelism on a multicomputer," in *ACM SIGPLAN Notices*, vol. 28, pp. 13–22, ACM, 1993.

[6] H. Printz, H. Kung, T. Mummert, and P. Scherer, "Automatic mapping of large signal processing systems to a parallel machine," in *33rd Annual Technical Symposium*, pp. 2–16, International Society for Optics and Photonics, 1989.

[7] U. K. Banerjee, *Dependence analysis for supercomputing.* Kluwer Academic Publishers, 1988.

[8] G. Goff, K. Kennedy, and C.-W. Tseng, *Practical dependence testing*, vol. 26. ACM, 1991.

[9] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, 1986.

[10] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A method for scheduling parallel loops," *Communications of the ACM*, vol. 35, no. 8, pp. 90–101, 1992.

[11] M. Chandy, K. Kennedy, C. Koelbel, C.-W. Tseng, *et al.*, "Integrated support for task and data parallelism," *International Journal of High Performance Computing Applications*, vol. 8, no. 2, pp. 80–98, 1994.

[12] J. J. Dongarra and D. C. Sorensen, "A portable environment for developing parallel fortran programs," *Parallel Computing*, vol. 5, no. 1, pp. 175–186, 1987.

[13] P. A. Suhler, J. Biswas, K. M. Korner, and J. C. Browne, "Tdfl: A task-level data flow language," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 103–115, 1990.

[14] R. G. Babb II and D. C. DiNucci, "Scientific parallel processing with lgdf2," in *Proceedings of the third SIAM Conference on Parallel Processing for Scientific Computing*, pp. 307–311, Society for Industrial and Applied Mathematics, 1987.

[15] R. Chandra, A. Gupta, and J. L. Hennessy, *COOL: A language for parallel programming.* Computer Systems Laboratory, Stanford University, 1989.

[16] S. Huynh, *Exploiting task-level parallelism automatically using pTask.* University of Toronto, 1996.

[17] D. Scales, M. Rinard, M. Lam, and J. Anderson, "Hierarchical concurrency in jade," in *Languages and Compilers for Parallel Computing*, pp. 50–64, Springer, 1992.

[18] M. S. Lam and M. C. Rinard, "Coarse-grain parallel programming in jade," in *ACM SIGPLAN Notices*, vol. 26, pp. 94–105, ACM, 1991.

[19] M.-Y. Wu and D. D. Gajski, "A programming aid for hypercube architectures," *The journal of Supercomputing*, vol. 2, no. 3, pp. 349–372, 1988.

[20] T. Yang and A. Gerasoulis, "Pyrros: static task scheduling and code generation for message passing multiprocessors," in *Proceedings of the 6th international conference on Supercomputing*, pp. 428–437, ACM, 1992.

[21] T. Gross, D. R. O'Hallaron, and J. Subhlok, "Task parallelism in a high performance fortran framework," *IEEE Concurrency*, vol. 2, no. 3, pp. 16–26, 1994.